LA-UR-21-32147
Approved for public release; distribution is unlimited.

Title: Charliecloud 101

Author(s): Priedhorsky, Reid
Ogas, Jordan Andrew
Easterday, Hunter Patrick

Intended for: sharing with external workshop organizers and attendees

Issued: 2021-12-13

Charliecloud 101

Thursday, January 14, 2021 / 12:30–4:30 pm
Reid Priedhorsky, Jordan Ogas, Hunter Easterday

1 Getting started

1.1 Description

This workshop will provide participants with background and hands-on experience to use basic containers for HPC applications. We will discuss what containers are, why they matter for HPC, and how they work. We'll give an overview of Charliecloud, the unprivileged container solution from HPC Division, and walk participants through installing it on their own compute resource. Participants will build toy containers and a real HPC application, and then run them in parallel on an HPC Division cluster. This will be a highly interactive workshop with lots of Q&A.

1.2 Prerequisites

1. Viewing Google Meet shared content. LANL G-Suite account preferred.
2. Join Mattermost channel.
3. Account on `gitlab.lanl.gov` (use previously provided instructions).
4. Laptop or workstation with:
   - SSH access to Grizzly (account has been created for you); and
   - Access to `gitlab.lanl.gov` on port 5050 **or** access to internet without going through the LANL web proxy.
   - SSH access to a Linux box with (Darwin satisfies these requirements):
     - x86_64 architecture
     - user namespaces enabled
     - access to the internet
     - C99 compiler
     - Python 3.5+
     - Python modules "`lark-parser`" and "`requests`"

1.3 Schedule (tentative)

12:15 –	*12:30*		*Teaching staff setup, last-minute Q&A*
12:30 –	1:15	45	Introduction to containers and Charliecloud (slides)
1:15 –	1:30	15	3. Key workflow operation: Pull
1:30 –	1:45	15	4. Containers are not special, part I: CentOS 7 via tarball
1:45 –	*2:00*	*15*	*break & catch-up*
2:00 –	2:30	30	5. Key workflow operation: Build from Dockerfile
2:30 –	2:55	25	6. Key workflow operation: Push
2:55 –	3:00	25	7a. MPI Hello World: Start pull
3:00 –	*3:15*	*15*	*break & catch-up; wait for pull*
3:15 –	3:35	20	7b. MPI Hello World: Build & run
3:35 –	4:05	30	8. TensorFlow
4:05 –	4:30	25	Closing and Q&A

2 Pre-workshop setup checklist

We strongly recommend you complete these preparatory steps before the workshop. If you have any trouble, let us know and we will be happy to help.

2.1 Join the Mattermost channel

You should have received an invite and a pointer to the correct channel.

2.2 Check SSH

You must be able to SSH:

- from your laptop to your Linux box
- from your laptop to `wtrw.lanl.gov`, and then on to `gr-fe`
- from your Linux box to `wtrw.lanl.gov`, and then on to `gr-fe`

2.3 Check your shell configuration

This command verifies that you do not have any Python configuration that will confuse Charliecloud. It should give no output.

```
$ env | fgrep PYTHON
```

2.4 Check container registry access

From the Linux box you intend to use, you must be able reach either the Docker Hub container registry or the registry on `gitlab.lanl.gov`. Docker Hub has a rather strict usage limit shared among all LANL users (because of the web proxy), so if you are on campus or using the VPN, we recommend you use `gitlab.lanl` (we have mirrored the necessary images); on the other hand, `gitlab.lanl` is *only* available on campus or through the VPN.

One way to test is below; at least one of these must report “Connection to ... succeeded”.

```
$ nc -vz -w5 gitlab.lanl.gov 5050
$ nc -vz -w5 registry-1.docker.io 443
```

2.5 Configure environment variables

Each shell on your Linux box (but not Grizzly) should have the following environment variables set. You can download `setup.sh` from Mattermost, edit it, and source it, as a shortcut.

Use our `ch-image` builder rather than Docker or whatever else you have installed:

```
$ export CH_BUILDER=ch-image
```

Configure Charliecloud paths. These can be changed to your preference; however, if you are on a shared resource like Darwin, use a location unique to you.

```
$ export CHORKSHOP=$HOME/chorkshop
$ export PREFIX=$CHORKSHOP/opt
$ export PATH=$PREFIX/bin:$PATH
```

2.6 Install Charliecloud

Charliecloud has a fairly standard Autotools build and is hopefully easy to build and install. We’ll use a pre-release because it has some new features we want.

Tip: We strongly recommend you type out most of the commands in this manual, rather than copying and pasting, because then they will pass through your brain and you will learn more.

```
$ mkdir $CHORKSHOP
$ cd $CHORKSHOP
$ wget https://github.com/hpc/charliecloud/releases/download/v0.22-
pre/charliecloud-0.22.pre+8e65fec.tar.gz
$ tar xf charliecloud-0.22.pre+8e65fec.tar.gz
$ cd charliecloud-0.22~pre+8e65fec
$ ./configure --prefix=$PREFIX
[...]
  with ch-image(1): yes
    enabled ... yes
    Python shebang line ... /usr/bin/env python3
    Python in shebang ≥ 3.4 ... ok (3.7.3)
    "lark-parser" module ... yes
    "requests" module ... yes
    ch-run(1) ... yes
[...]
  ch-run(1): yes
    user+mount namespaces ... yes
[...]
```

Check the configure report and verify that the blue text above matches your system. This is important — you will not be able to follow along if it doesn't!

Common problem 1: If `lark-parser` and `requests` are missing, you can install them into your home directory (under `~/.local`) with the following. Do so and run `./configure` again.

```
$ pip3 install lark-parser --user
$ pip3 install requests --user
```

Common problem 2: If "`user+mount namespaces`" reports "`no`", this must be fixed by your sysadmin. Check with us and we'll help you fix it, either by supplying you with commands to fix it or helping you find a box that works.

Build and install:

```
$ make
$ make install
$ which ch-run
/home/reidpr/chorkshop/opt/bin/ch-run
$ ch-run --version
0.22~pre+8e65fec
$ ch-image --dependencies  # should give no output
$ ch-image --version
0.22~pre+8e65fec
```

If you prefer, you can run Charliecloud directly from the build directory, without installing it. In this case, skip "`make install`", and add `$CHORKSHOP/charliecloud-0.22~pre+8e65fec/bin` to your `$PATH`.

3 Key workflow operation: Pull

To start, let's obtain a container image that someone else has already built. The containery way to do this is the *pull* operation, which means to move an image from a remote repository into local storage of some kind.

First, let's browse the Docker Hub repository of official CentOS images.[1] Note the list of *tags*; this is a partial list of image versions that are available. We'll use the tag "7".

Use the Charliecloud program `ch-image` to pull this image to a directory.

> *Note:* The examples assume that you are using the images on `gitlab.lanl.gov`. If you are using the ones on Docker Hub, remove "`gitlab.lanl.gov:5050/reidpr/charliecloud/`" from the image references.

```
$ ch-image --help
usage: ch-image [-h] [--dependencies] [--no-cache] [-s DIR] [--tls-no-verify]
                [-v] [--version]
                CMD ...

Build and manage images; completely unprivileged.
[...]
$ ch-image pull --help
usage: ch-image pull [-h] [--last-layer N] [--parse-only]
                     IMAGE_REF [IMAGE_DIR]
[...]
$ cd $CHORKSHOP
$ ch-image pull gitlab.lanl.gov:5050/reidpr/charliecloud/centos:7 ./centos:7
pulling image:   gitlab.lanl.gov:5050/reidpr/charliecloud/centos:7
destination:     ./centos:7/
manifest: downloading
layer 1/1: 2d473b0: downloading
layer 1/1: 2d473b0: listing
validating tarball members
resolving whiteouts
flattening image
layer 1/1: 2d473b0: extracting
done
```

Examine the image. It looks like the root directory of a standard Linux distribution.

```
$ ls ./centos:7
anaconda-post.log  dev  home  lib64  mnt  proc  run   srv  tmp  var
bin                etc  lib   media  opt  root  sbin  sys  usr
```

Images can come in lots of different formats. This one is just a directory, which is the format that ch-run needs.

[1] https://hub.docker.com/_/centos

Run a container:

> *Tip:* Terminal activity in a container is written in blue.

```
$ ch-run ./centos:7 -- /bin/bash
$ pwd
/
$ ls
anaconda-post.log  dev  home  lib64  mnt  proc  run   srv  tmp  var
bin                etc  lib   media  opt  root  sbin  sys  usr
$ cat /etc/redhat-release
CentOS Linux release 7.9.2009 (Core)
$ exit
```

What does this command do?

1. Start a container (`ch-run`).
2. Use the image in directory `$CHORKSHOP/centos:7`.
3. Stop processing ch-run command line arguments (`--`). (Note this is standard notation for UNIX apps.)
4. Run the program `/bin/bash` inside the container, which starts an interactive shell where we enter a few commands and then exit, returning to the host.

4 Containers are not special, part I: CentOS 7 via tarball

Many folks would like you to believe that containers are magic and special. **This is not the case.** To demonstrate, we'll create a working container image using standard UNIX tools. (Another angle on this is given in §10 below.)

CentOS provides a tarball containing an installed CentOS 7 base image; we can use that in Charliecloud directly. (In fact, this tarball is what's used to create the image in the previous section.)

```
$ cd $CHORKSHOP
$ wget -O centos.tar.xz 'https://github.com/CentOS/sig-cloud-instance-
images/raw/CentOS-7-x86_64/docker/centos-7-x86_64-docker.tar.xz?raw=true'
$ tar tf centos.tar.xz | head
./
./dev/
./proc/
./run/
./run/lock/
./run/lock/lockdev/
./run/lock/subsys/
./run/cryptsetup/
./run/utmp
./run/systemd/
```

This tarball is what's called a "tarbomb", so we need to provide an enclosing directory to avoid making a mess.

```
$ mkdir centos
$ cd centos
$ tar xf ../centos.tar.xz
$ ls
anaconda-post.log  dev  home  lib64  mnt  proc  run   srv  tmp  var
bin                etc  lib   media  opt  root  sbin  sys  usr
$ cd -
```

Now, run Bash in the container!

```
$ ch-run ./centos -- /bin/bash
$ pwd
/
$ ls
anaconda-post.log  dev  home  lib64  mnt  proc  run   srv  tmp  var
bin                etc  lib   media  opt  root  sbin  sys  usr
$ cat /etc/redhat-release
CentOS Linux release 7.9.2009 (Core)
$ exit
```

Note: CentOS distributes tarballs with some odd directory permissions that make them un-deleteable. To remove this directory:

```
$ chmod -R u+w ./centos
$ rm -Rf --one-file-system ./centos
```

5 Key workflow operation: Build from Dockerfile

The other containery way to get an image is the *build* operation. This interprets a recipe, usually a *Dockerfile*, to create an image and place it into *builder storage*. We can then extract the image from builder storage to a directory and run it.

5.1 Exercise

We'll write a "Hello World" Python program and run it within a container we specify with a Dockerfile. Set up a directory to work in:

```
$ cd $CHORKSHOP
$ mkdir hello.src
$ cd hello.src
```

Type in the following program as "`hello.py`" using your least favorite editor.

```
#!/usr/bin/python3

print("Hello World!")
```

Next, create a file called "`Dockerfile`" and type in the following 4-line recipe:

```
FROM gitlab.lanl.gov:5050/reidpr/charliecloud/centos:7
RUN yum -y install python36
COPY ./hello.py /
RUN chmod 755 /hello.py
```

These four instructions say:

1. We are extending the `centos:7` *base image*. (Note that this is a different instance of CentOS 7 than we downloaded above.)
2. Install the `python36` RPM package, which we need for our Hello World program.
3. Copy the file `hello.py` we just made to the root directory of the image. In the source argument, the path is relative to the *context directory*, which we'll see more of below.
4. Make that file executable.

Let's build the image:

```
$ ls
Dockerfile  hello.py
$ ch-build --builder-info
builder: ch-image: 0.21
$ ch-build -t hello -f Dockerfile .
```

Charliecloud supports multiple *builders*. In this workshop, we are using `ch-image`, which comes with Charliecloud, but you can also use Docker or Buildah. The wrapper script `ch-build` provides a unified interface to their basic functionality.

> *Note:* `ch-image` is a big deal because it is completely unprivileged, which is important in environments like ours. Other builders run as root or require setuid root helper programs; this raises a number of security questions. (Buildah does have a fully unprivileged mode that we contributed.)

The `ch-build` line says:

1. Build an image named (*tagged*) "hello".
2. Use the Dockerfile called "Dockerfile".
3. Use the current directory as the context directory.

Now list the images ch-image knows about:

```
$ ch-image list
gitlab.lanl.gov:5050/reidpr/charliecloud/centos:7
hello
```

Extract the image to a directory. Make sure you get "`unpacked ok`".

```
$ cd $CHORKSHOP
$ ch-builder2tar hello .
builder: ch-image
exporting
 344MiB 0:00:00 [ <=>                                        ]
compressing
 344MiB 0:00:01 [============================================>] 100%
-rw-r----- 1 reidpr reidpr 117M Apr  7 10:47 ./hello.tar.gz
$ ch-tar2dir hello.tar.gz .
creating new image ./hello
 116MiB 0:00:02 [============================================>] 100%
./hello unpacked ok
```

And run it:

```
$ ch-run ./hello -- /hello.py
Hello World!
```

Note that we've run our application directly rather than starting an interactive shell.

5.2 Further reading

- Dockerfile reference:
  https://docs.docker.com/engine/reference/builder/

6 Key workflow operation: Push

The containery way to share your images is by *pushing* them to a container registry. (Above, we did the reverse of this operation: pulling from a registry.) In this section, we will set up a registry on `gitlab.lanl.gov` and push the hello image to that registry, then pull it back to compare.

6.1 Set up registry

Create a private container registry:

1. Browse to `https://gitlab.lanl.gov`.
2. Log in with *LANL Weblogin*. You should end up on your *Projects* page.
3. Click *New project*.
4. Name your project "`ws2021-01`". Leave *Visibility Level* at *Private*. Click *Create project*. You should end up at your project's main page.
5. At left, choose *Settings* (the gear icon) → *General*, then Visibility, project features, permissions. Enable *Container registry*, then click *Save changes*.
6. At left, choose *Packages & Registries* (the box icon) → *Container registry*. You should see the message "*There are no container images stored for this project*".

At this point, we have a container registry set up, and we need to teach `ch-image` how to log into it. If you have a CryptoCard, you may be able to use that. However, GitLab has a thing called a *personal access token* (PAT) that can be used no matter how you log into the GitLab web app. To create one:

7. Click on your avatar at the top right. Choose *Settings*.
8. At left, choose *Access Tokens* (the three-pin plug icon).
9. Type in the name "`registry`". Tick the boxes *read_registry* and *write_registry*. Click *Create personal access token*.
10. Your PAT will be displayed at the top of the result page under *Your new personal access token*. Copy this string and store it somewhere safe & policy compliant. (Also, you can revoke it at the end of the workshop if you like.)

6.2 Push image

We can use "`ch-image push`" to push the image to `gitlab.lanl`.

```
$ cd $CHORKSHOP
$ ch-image list
gitlab.lanl.gov:5050/reidpr/charliecloud/centos:7
hello
$ ch-image push --help
usage: ch-image push [-h] [--image DIR] IMAGE_REF [DEST_REF]

push image from local filesystem to remote repository
[...]
```

Note that the tagging step you would need for Docker is unnecessary here, because we can just specify a destination reference at push time.

When you are prompted for credentials, enter your e-mail address (that you use to log into `gitlab.lanl.gov`) and copy-paste the PAT you created earlier. (Currently you will be prompted multiple times, which is a known bug.)

```
$ ch-image push hello gitlab.lanl.gov:5050/reidpr/ws2021-01/hello:latest
pushing image:   hello
destination:     gitlab.lanl.gov:5050/reidpr/ws2021-01/hello:latest
layer 1/1: gathering
warning: stripping unsafe setuid bit: ./usr/bin/chage
warning: stripping unsafe setuid bit: ./usr/bin/chfn
[...]
layer 1/1: preparing
preparing metadata
```

```
starting upload
layer 1/1: bca515d: checking if already in repository
anonymous access rejected

Username: reidpr@lanl.gov
Password:
layer 1/1: bca515d: not present, uploading
anonymous access rejected

Username: reidpr@lanl.gov
Password:
config: f969909: checking if already in repository
config: f969909: not present, uploading
manifest: uploading
cleaning up
done
```

Note: Upload can be slow, so be patient. There is no progress bar yet.

Go back to your container registry page. You should see your image listed now!

6.3 Pull and compare

Let's pull that image and see how it looks.

```
$ ch-image pull gitlab.lanl.gov:5050/reidpr/ws2021-01/hello:latest ./hello.2
pulling image:   gitlab.lanl.gov:5050/reidpr/ws2021-01/hello:latest
destination:     hello.2
[...]
$ ls hello
anaconda-post.log  dev       home   media   proc   sbin   tmp   WEIRD_AL_YANKOVIC
bin                etc       lib    mnt     root   srv    usr
ch                 hello.py  lib64  opt     run    sys    var
$ ls hello.2
anaconda-post.log  ch   etc       home  lib64  mnt  proc  run   srv  tmp  var
bin                dev  hello.py  lib   media  opt  root  sbin  sys  usr
$ diff -ur --no-dereference hello hello.2
Only in hello: WEIRD_AL_YANKOVIC
$ diff -u <(cd hello && ls -R) <(cd hello.2 && ls -R)
--- /dev/fd/63     2021-01-12 17:44:17.832325448 -0700
+++ /dev/fd/62     2021-01-12 17:44:17.832325448 -0700
@@ -20,7 +20,6 @@
 tmp
 usr
 var
-WEIRD_AL_YANKOVIC

 ./ch:
 environment
```

7 MPI Hello World

The next exercise demonstrates a typical workflow of:

1. Build image locally.
2. Copy image tarball to HPC cluster.
3. Run application on HPC cluster.

We'll use a simple parallel application. The base image is a CentOS 8 image with OpenMPI already installed; OpenMPI takes about 30 minutes to build and install, so we don't want to take workshop time doing that.

7.1 Pull base image

This step is not strictly necessary, because "`ch-image build`" will pull the image if needed, but this particular image is quite large, so it may be useful to start the pull and then go have a break. (Recall that `ch-image` does not have progress bars yes; be patient.)

> *Note:* The reference for the base image is "`charliecloud/openmpi`" on Docker Hub, not plain "`openmpi`", because we uploaded it there.

```
$ ch-image pull gitlab.lanl.gov:5050/reidpr/charliecloud/openmpi
```

7.2 Build image

Create a new directory for this project, and within it the following simple C program. (Note the program contains a bug; consider fixing it.)

```
$ cd $CHORKSHOP
$ mkdir mpihello
$ cd mpihello
$ vim mpihello.c  # or download from Mattermost
$ cat mpihello.c
#include <stdio.h>
#include <mpi.h>

int main (int argc, char **argv)
{
   int msg, rank, rank_ct;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &rank_ct);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   printf("hello from rank %d of %d\n", rank, rank_ct);

   if (rank == 0) {
      for (int i = 1; i < rank_ct; i++) {
         MPI_Send(&msg, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
         printf("rank %d sent %d to rank %d\n", rank, msg, i);
      }
   } else {
      MPI_Recv(&msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("rank %d received %d from rank 0\n", rank, msg);
   }

   MPI_Finalize();
}
```

Add the following Dockerfile.

```
$ vim Dockerfile
$ cat Dockerfile
FROM gitlab.lanl.gov:5050/reidpr/charliecloud/openmpi
#FROM charliecloud/openmpi

RUN mkdir hello
WORKDIR hello
COPY mpihello.c .
RUN mpicc -o mpihello mpihello.c
```

The instruction `WORKDIR` changes directories (the default working directory within a Dockerfile is `/`).

Build:

```
$ ls
Dockerfile  mpihello.c
$ ch-build -t mpihello .
```

Note that the default Dockerfile is `./Dockerfile`; we can omit `-f`.

7.3 Copy to HPC

Next, we obtain an image tarball and copy it to our Turquoise home directory.

```
$ cd $CHORKSHOP
$ ch-builder2tar mpihello .
$ scp mpihello.tar.gz wtrw:gr-fe:~
mpihello.tar.gz                              100%  259MB  93.9MB/s   00:02
```

7.4 Log into Grizzly

In a new terminal:

```
$ ssh wtrw.lanl.gov
$ ssh gr-fe
```

7.5 Run application

We'll run this application interactively. One could also put similar steps in a Slurm batch script.

First, obtain a two-node allocation and load the Charliecloud module. (You may notice that the version is a little behind; that's not a problem for our current purposes.)

```
$ salloc -N1 -t 3:00:00 --reservation=CLASS-193912
salloc: Granted job allocation 599518
[...]
$ module load charliecloud
$ ch-run --version
0.20
```

Unpack the image into the tmpfs (RAM disk) on both nodes. Make sure you get *two* "`unpacked ok`".

```
$ srun ch-tar2dir ~/mpihello.tar.gz /var/tmp
creating new image /var/tmp/mpihello
creating new image /var/tmp/mpihello
/var/tmp/mpihello unpacked ok
/var/tmp/mpihello unpacked ok
```

Run the application on all 72 cores in your allocation:

```
$ srun -c1 ch-run --join /var/tmp/mpihello -- /hello/mpihello
hello from rank 22 of 72
rank 22 received 0 from rank 0
hello from rank 14 of 72
rank 14 received 0 from rank 0
[...]
hello from rank 0 of 72
rank 0 sent 0 to rank 1
rank 0 sent 0 to rank 2
[...]
hello from rank 65 of 72
rank 65 received 0 from rank 0
```

Win!

Why `--join`? By default, each containerized rank is in a different container, and processes in sibling containers can't attach to one another to do the kind of shared memory that OpenMPI prefers. Sometimes this fails, and sometimes it's just slower. By adding `--join`, the independent `ch-run` invocations use the same container.

Leave the Slurm allocation running. We'll use it in the next exercise too.

8 TensorFlow

In the final exercise, we'll run a TensorFlow image on Grizzly. This image is quite large, so we'll use a tarball pre-staged on the clusters.

Currently, we could not pull the image from a repository on a Turquoise cluster, because both Docker Hub and `gitlab.lanl.gov` are blocked by our firewall, and the compute nodes have no internet access at all.

8.1 Set up SSH tunnels

We will use a local web browser to access the Jupyter Notebook kernel offered by the image. This requires setting up a three-hop SSH tunnel. In a new terminal on your laptop, do the following. `grZZZZ` is the first node in your Slurm allocation from above, which is the one your shell is on. *`X`* and *`Y`* are arbitrary numbers in the range 1024–65535 **that no one else is using.**

```
$ ssh -S none -L 8888:localhost:X wtrw.lanl.gov
$ ssh -L X:localhost:Y gr-fe
$ ssh -L Y:localhost:8888 grZZZZ
```

If you get an error about can't bind to port, use a different number. "`-S none`" means to not use an existing multiplex connection, if you have one; this is needed because you can't add port forwarding to an existing connection in this way.

8.2 Unpack and start the application

This exercise uses only a single node, so we don't need to `srun` unpacking. Also, we can unpack the tarball directly from project space without copying it. In your Grizzly allocation still running from above:

```
$ ch-tar2dir flow.tar.gz /var/tmp
[...]
$ ch-run --write --cd=/tf /var/tmp/flow -- jupyter notebook
[I 23:12:41.826 NotebookApp] Serving notebooks from local directory: /tf
[...]
    To access the notebook, open this file in a browser:
        file:///home/reidpr/.local/share/jupyter/runtime/nbserver-1955-
open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=[...]
```

There are two new flags here. `--write` makes the container image writeable; this is not a best practice, but we use it here for expediency (do as I say, not as I do). `--cd` sets the working directory when the container starts.

This will emit a `localhost` URL. Paste this URL into a local browser. Jupyter thinks that `localhost` refers to the Grizzly node, but because of our tunnel, the same URL will also work from your laptop.

8.3 Run the notebook?

In your browser:

1. Navigate to *tensorflow-tutorials* → *classification.ipynb*.
2. Select *Cell* → *All Output* → *Clear*
3. Select *Cell* → *Run All*.
4. Scroll down to cell 5.
5. Note how the cell has exploded all over with a big stack trace.

This illustrates how containerization is not magic. You can't simply pull arbitrary containers off the internet, even high-reputation ones like TensorFlow, and be confident they will work. Best practices are important.

In this case, the container assumes (a) it can write to its image and (b) it has arbitrary access to the internet at runtime. We work around (a) with `--write`, but (b) is a cardinal sin of HPC containers. Note how the assumption is so deeply embedded that there is no proper error message and no instructions on how to load the data if it can't be downloaded. The way we debugged this was some painful source code examination and googling; it turns out TensorFlow has a cache directory in your home directory that you can pre-populate, which is how we'll work around this issue.

(Some of the other TensorFlow demos are even worse — they also assume they can shell out and run `pip install`, which violates the best practice of using a clear and complete build recipe.)

8.4 Manually add data to cache

In your second Grizzly terminal:

```
$ mkdir -p ~/.keras/datasets
$ cp -r /usr/projects/charliecloud/public/fashion-mnist ~/.keras/datasets
$ ls -lh ~/.keras/datasets/fashion-mnist/
total 30M
-rw-rw---- 1 reidpr reidpr 4.3M Jan 13 15:17 t10k-images-idx3-ubyte.gz
-rw-rw---- 1 reidpr reidpr 5.1K Jan 13 15:17 t10k-labels-idx1-ubyte.gz
-rw-rw---- 1 reidpr reidpr  26M Jan 13 15:17 train-images-idx3-ubyte.gz
-rw-rw---- 1 reidpr reidpr  29K Jan 13 15:17 train-labels-idx1-ubyte.gz
```

Now the notebook will use these gzip files instead of trying to download them. (The specific destination directory is important!)

8.5 Run the notebook again

Go back to the top of the notebook. Instead of running all the cells together, run each individually with Shift-Enter so you can watch what happens. We'll go through Cell 17, which shows a progress report as the model trains. Others below show plots and some pictures as the ~~statistics~~ ~~machine learning~~ ~~deep learning~~ artificial intelligence happens.

9 *Appendix:* Namespaces with `unshare(1)`

Note: This appendix will not be covered in the workshop but contains some background material that we found interesting and informative.

`unshare(1)` is a shell command that comes with most new-ish Linux distributions in the `util-linux` package. We will use it to explore a little about how namespaces, which are the basis of containers, work.

9.1 Exercise 1: Identifying namespaces

Namespaces form a tree, and every process is already in all namespaces. Every namespace has an ID number, which you can see in `/proc` with some magic symlinks:

```
$ cd $CHORKSHOP
$ ls -l /proc/self/ns | tee outside.txt
total 0
lrwxrwxrwx 1 reidpr reidpr 0 Mar 31 16:44 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 reidpr reidpr 0 Mar 31 16:44 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 reidpr reidpr 0 Mar 31 16:44 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx 1 reidpr reidpr 0 Mar 31 16:44 net -> 'net:[4026531992]'
lrwxrwxrwx 1 reidpr reidpr 0 Mar 31 16:44 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 reidpr reidpr 0 Mar 31 16:44 pid_for_children ->
'pid:[4026531836]'
lrwxrwxrwx 1 reidpr reidpr 0 Mar 31 16:44 user -> 'user:[4026531837]'
lrwxrwxrwx 1 reidpr reidpr 0 Mar 31 16:44 uts -> 'uts:[4026531838]'
```

Let's start a new shell with different namespaces. Note how the ID numbers change.

```
$ unshare --user --mount
$ ls -l /proc/self/ns | inside.txt
total 0
lrwxrwxrwx 1 nobody nogroup 0 Mar 31 16:46 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 nobody nogroup 0 Mar 31 16:46 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 nobody nogroup 0 Mar 31 16:46 mnt -> 'mnt:[4026532733]'
lrwxrwxrwx 1 nobody nogroup 0 Mar 31 16:46 net -> 'net:[4026531992]'
lrwxrwxrwx 1 nobody nogroup 0 Mar 31 16:46 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 nobody nogroup 0 Mar 31 16:46 pid_for_children ->
'pid:[4026531836]'
lrwxrwxrwx 1 nobody nogroup 0 Mar 31 16:46 user -> 'user:[4026532732]'
lrwxrwxrwx 1 nobody nogroup 0 Mar 31 16:46 uts -> 'uts:[4026531838]'
$ exit
```

9.2 Exercise 2: The user namespace

Unprivileged user namespaces let you map your effective UID to any UID inside the namespace, and your effective GID to any GID. Let's try it. First, who are we:

```
$ id
uid=1000(reidpr) gid=1000(reidpr)
groups=1000(reidpr),24(cdrom),25(floppy),27(sudo),29(audio)
```

This shows our user (1000/`reidpr`), our primary group (1000/`reidpr`), and a bunch of supplementary groups.

Let's start a user namespace, mapping our UID to 0/`root` and my GID to 0/`root`. (Older versions of `unshare` do not let you specify the mappings directly.)

```
$ unshare --user --map-root-user
# id
uid=0(root) gid=0(root) groups=0(root),65534(nogroup)
```

This shows that our UID is 0, our GID is 0, and all the supplementary groups have collapsed into 65534/`nogroup`, because they are unmapped inside the namespace. (If `id` complains about not finding names for IDs, just ignore it.)

We are root!!! Let's try something sneaky!!!

```
# cat /etc/shadow
cat: /etc/shadow: Permission denied
```

Drat! The kernel followed the UID map outside the namespace and used that for access control; i.e., we are still acting as ourselves, a normal unprivileged user. Something else interesting:

```
# ls -l /etc/shadow
-rw-r----- 1 nobody nogroup 2151 Feb 10 11:51 /etc/shadow
# exit
```

This shows up as `nobody:nogroup` because UID 0 and GID 0 on the outside are unmapped.

9.3 Exercise 3: The mount namespace

This namespace lets us set up an independent filesystem tree. For this exercise, you will need two terminals.

In Terminal 1, set up namespaces and mount a new tmpfs over your home directory.

```
$ unshare --mount
unshare: unshare failed: Operation not permitted
```

Wait! What!? The problem is that mount is a privileged namespace. We need to add the user namespace to make it an unprivileged operation. Try again:

```
$ unshare --mount --user
$ mount -t tmpfs none /home/reidpr
mount: only root can use "--types" option
```

Wait! What!? The problem now is that you still need to be root inside the container to use the `mount(2)` system call. Try again:

```
$ unshare --mount --user --map-root-user
# mount -t tmpfs none /home/reidpr
# mount | fgrep /home/reidpr
none on /home/reidpr type tmpfs (rw,relatime,uid=1000,gid=1000)
# touch /home/reidpr/foo
# ls /home/reidpr
foo
```

In Terminal 2, which is not in the container, note how the mount does not show up in `mount` output and the files you created are not present:

```
$ ls /home/reidpr
articles.txt            flu-index.tsv          perms_test
[...]
$ mount | fgrep /home/reidpr
$
```

Exit the container in Terminal 1:

```
# exit
```

9.4 Further reading

- `unshare(1)` man page (note this is the most current version; yours may differ): http://man7.org/linux/man-pages/man1/unshare.1.html
- `namespaces(7)` man page: http://man7.org/linux/man-pages/man7/namespaces.7.html
- *Linux Weekly News* article series on namespaces: https://lwn.net/Articles/531114/

10 *Appendix:* All you need is Bash

Note: This appendix will not be covered in the workshop but contains some background material that we found interesting and informative.

In this exercise, we'll use shell commands to create minimal container image with a working copy of Bash, and that's it. To do so, we need to set up a directory with the Bash binary, the shared libraries it uses, and a few other hooks needed by Charliecloud.

Important: Your Bash is probably linked differently than described below. Use the paths from your terminal, not the workshop manual. **Adjust the steps below as needed.** It will not work otherwise!

```
$ ldd /bin/bash
    linux-vdso.so.1 (0x00007ffdafff2000)
    libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007f6935cb6000)
    libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f6935cb1000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f6935af0000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f6935e21000)
$ ls -l /lib/x86_64-linux-gnu/libc.so.6
lrwxrwxrwx 1 root root 12 May  1  2019 /lib/x86_64-linux-gnu/libc.so.6 ->
libc-2.28.so
```

The shared libraries pointed to are symlinks, so we'll use `cp -L` to dereference them and copy the target files. Note that `linux-vdso.so.1` is a kernel thing, not a shared library file.

Set up the container:

```
$ mkdir $CHORKSHOP/alluneed
$ cd $CHORKSHOP/alluneed
$ mkdir bin
$ mkdir dev
$ mkdir lib
$ mkdir lib64
$ mkdir lib/x86_64-linux-gnu
$ mkdir proc
$ mkdir sys
$ mkdir tmp
$ cp -pL /bin/bash ./bin
$ cp -pL /lib/x86_64-linux-gnu/libtinfo.so.6 ./lib/x86_64-linux-gnu
$ cp -pL /lib/x86_64-linux-gnu/libdl.so.2 ./lib/x86_64-linux-gnu
$ cp -pL /lib/x86_64-linux-gnu/libc.so.6 ./lib/x86_64-linux-gnu
$ cp -pL /lib64/ld-linux-x86-64.so.2 ./lib64/ld-linux-x86-64.so.2
$ cd $CHORKSHOP
$ ls -lR alluneed
./alluneed:
total 0
drwxr-x--- 2 reidpr reidpr 60 Mar 31 17:15 bin
drwxr-x--- 2 reidpr reidpr 40 Mar 31 17:26 dev
drwxr-x--- 2 reidpr reidpr 80 Mar 31 17:27 etc
drwxr-x--- 3 reidpr reidpr 60 Mar 31 17:17 lib
drwxr-x--- 2 reidpr reidpr 60 Mar 31 17:19 lib64
drwxr-x--- 2 reidpr reidpr 40 Mar 31 17:26 proc
drwxr-x--- 2 reidpr reidpr 40 Mar 31 17:26 sys
drwxr-x--- 2 reidpr reidpr 40 Mar 31 17:27 tmp

./alluneed/bin:
total 1144
-rwxr-xr-x 1 reidpr reidpr 1168776 Apr 17  2019 bash

./alluneed/dev:
total 0

./alluneed/lib:
```

```
total 0
drwxr-x--- 2 reidpr reidpr 100 Mar 31 17:19 x86_64-linux-gnu

./alluneed/lib/x86_64-linux-gnu:
total 1980
-rwxr-xr-x 1 reidpr reidpr 1824496 May  1  2019 libc.so.6
-rw-r--r-- 1 reidpr reidpr   14592 May  1  2019 libdl.so.2
-rw-r--r-- 1 reidpr reidpr  183528 Nov  2 12:16 libtinfo.so.6

./alluneed/lib64:
total 164
-rwxr-xr-x 1 reidpr reidpr 165632 May  1  2019 ld-linux-x86-64.so.2

./alluneed/proc:
total 0

./alluneed/sys:
total 0

./alluneed/tmp:
total 0
```

Next, start a container and run `/bin/bash` within it. Options `--no-home` and `--no-passwd` turn off some convenience features that this image isn't prepared for.

```
$ ch-run --no-home --no-passwd /var/tmp/alluneed -- /bin/bash
$ pwd
/
$ echo "hello world"
hello world
$ ls /
bash: ls: command not found
$ echo *
bin dev home lib lib64 proc sys tmp
$ exit
```

It's not very useful since the only commands we have are Bash built-ins, but it's a container!